6. Finding Efficient Compressions; Huffman and Hu-Tucker

We now address the question: how do we find a code that uses the frequency information about k length patterns efficiently to shorten our message?

We have, for each of our blocks say, block q, a number of occurrences, f(q) and we want to assign a code word c(q). Let l(q) be the length of that word. The new want to minimize the sum over all q of the product c(q)*l(q).

The way to do this is remarkably simple. It is based upon the following two statements.

1. We
should, for efficiency, always give the least frequently occurring blocks the longest code words that we have.
2. There are, in any efficient code, at least two longest code words.

To see how these two statements provide us with an algorithm for producing a code from our frequency information, and to see where they come from, we first note how a code can be represented by a tree.

We will associate **a binary tree**
to a code, by assigning a set of code words to each node (or vertex) of the tree.

I look at our trees upside down. So the top node is the "root" of the tree, and to it we associate the entire set of blocks which occur at all in our message.

We give each vertex either 0 or two children. The children will correspond to the blocks that are assigned 0 and 1 at that node. (We might as well put the 0's on the left). We label our vertices with the label of its parent plus an additional bit of 0 if it is the left child or 1 if it is the right child of the parent.

Thus, the left child of the root corresponds to the set of the blocks that have 0 as their first bit, and the right child corresponds to those having 1 as first bit, and so on down the tree. The left child has label 0, the right child label 1.

The four vertices at the second level correspond to the sets of blocks that, in order, have first two bits given by 00, 01, 10, and 11 and we can label the vertices as such.

While we are at it, we might as well assign to each vertex the number of occurrences of the blocks that get code words that begin with its bits.

As we go down the tree, the blocks associated with each vertex become fewer and fewer, and eventually there is a single block associated with the word. We make that vertex a leaf of our code tree, so that it has no children.

If we assign a code word to an interior vertex of our tree, and there is some other code word that begins with it, we may have some trouble decoding our code. If we have all our code words leaves of the tree, then we can immediately recognize when a codeword ends, so that we can unambiguously decode any message. (This is so because no leaf is a prefix of any other leaf.)

So we only use code words that are leaves of our tree,

Now let us look at the longest code words. They will correspond to leaves at the bottom of our tree. Each has a parent, and each parent has two children, so that each word at the bottom of the tree has a partner there as well, which was our second statement.

To find our optimal code, we will work backwards from the bottom of the tree as follows.

Since the least frequent two blocks should be at the bottom level of the tree, and each leaf there has a sibling with the same parent, we might as well choose them as siblings. Which means we assign the last bit of one of them to be 0 and of the other to be 1, and require that their other bits be identical.

This simple act reduces the problem of creating an optimal code with N blocks, to one of creating one with N-1 blocks, where these two least frequent blocks are replaced by a new artificial block which occurs whenever either of them occurred, that is, with frequency that is the sum of theirs.

This is all we need to create the entire code. We repeat this step until there is only one (artificial) block left.

Consider the following example: suppose we had 8 blocks with frequencies 20,8,4,4,3,2,2,1,1,

First we apply this step to the two blocks with frequency 1. They are assigned (0,1) and their joint frequency is 2. We combine them next with the one of the other blocks with frequency 2 to produce one with frequency 4 whose code words end with (0,10,11)

Next we merge the other frequency 2 block with the frequency 3 block to get a frequency 5 block that internally is (0,1).

At this point our frequency sequence looks like: 20, 8,5(0,1), 4,4,4(0,10,11), where the numbers within each parenthesis describe the internal subtree for the artificial block whose frequency precedes that parenthesis.

Applying our step to the last two and then the next two produces the sequence

20, 9(0,10,11), 8, 8 (0,10,110,111)

Next the two 8's can be combined, then the resulting 16 and 9 and we have two remaining frequencies:

20, 25(00,010,011,10,110,1110,11110,11111).

Finally we merge these, and we have as our code words:

(0, 100,1010, 1011,110,1110,11110,111110,111111).

We can assign these to blocks by ordering them by their lengths, shortest first, and ordering the blocks by their frequencies, largest first, and pairing them.

Notice that 0 and 1 could be interchanged at each vertex without changing optimality, and it does not really matter here which block gets which code word among code words having the same length.

Actually, we do not have to keep track of the code word suffixes at each node as we did above; it is sufficient to keep track of the lengths of these suffixes, and we can easily recreate a code with codewords having the given lengths from that informaiton

This method for finding a code is called Huffman's algorithm after its discoverer, who was an MIT student who was given an assignment to find a good code based on frequency information alone..

It is so easy, that we try a harder problem just to keep you awake.

The Hu-Tucker Algorithm: Shortest Alphabetical Codes.

If the messages to be encoded are blocks of bits of length k, they have a natural ordering, which we can obtain by treating each bit string as a binary number and using the ordering of these numbers. In general the given messages may have some fixed order.

We now ask, how can we construct a code that has minimum total message length as before, but the ordering of the code words (where word x is smaller than word y when at the most significant bit where they differ x is smaller.) is required to be the same as the ordering of their corresponding blocks.

Thus, the block that is smaller in the given block ordering is to get the "lexicographically smaller" code word, which means the one representing a smaller binary number, with an assumed decimal point in front of each. (This is done so that the variable length of code words does not make longer words bigger than short ones. Thus we want 10 to be bigger than 01111.)

Of course blocks also have frequencies as before and we still want to minimize the length of the total message, subject to this restriction: the ordering of the code words as binary sequences following a decimal point must be the same as the initial ordering of the messages.

There is a neat algorithm for this problem, though it is somewhat strange, and not at all easy to prove that it works. (The first few published proofs were wrong.)

Here is how it works:

There are three steps: First you merge vertices together, just as in the Huffman algorithm, except that the merging rules are slightly different here...

Second, you use the resulting merge pattern only to determine the length of each code word.

Third you construct the final tree from these lengths.

How does the merging step differ here?

In the previous problem we merged the two blocks that have the smallest frequencies together.

Now we introduce a notion of compatibility among blocks. Again, we will have original blocks and artificial merged blocks.

The compatibility rule is:
you can only merge two blocks if there are no original blocks left between them. Thus if you have

three blocks in order with frequencies 2,4,3 you cannot merge the 2 and 3 frequency blocks together unless the 4 frequency block is artificial.

And here is the rule for merging**: if x is the lowest frequency block compatible to y and y is the lowest frequency block compatible to x , you should merge them**.

So here is how the algorithm goes: you merge blocks together according to this rule until they all merge into one.;
you keep track of the number of merges of each block, in order, which will be the lengths of the code words of the blocks,.

Then you construct an alphabetic tree having these lengths. There will be only one possible way to do this.

We have yet to describe how to perform this last step. But before doing so let us look at an example.

Suppose our frequencies are, in the order that we want to preserve:

1,2,23,4,3,3,5,19.

At this stage each block is compatible only with its immediate neighbor. The only pairs that obey our condition that x is the lowest compatible with y and vice versa are the first two and the 3,3 . we can merge each of these, getting

3(1,1), 23, 4, 6(1,1) 5, 19.

The notation here is
that what is in each parenthesis are the lengths of the suffixes of the words merged into that artificial block.

Now we can merge the the
3 and 23, and also the 4 and 5, which we can locate where the old 4 was:

26(2,2,1), 9(1,1),6(1,1),19

Here the restriction about compatibility no longer requires anything and merging is like it was in the Huffman algorithm.

Thus, next we can merge the 9 and 6 which gives (26(2,2,1),15(2,2,2,2),19), then the 15 and 19 and finally the whole thing together, getting 26(2,2,1),34(3,3,3,3,1) then 60(3,3,2,4,4,4,4,2).

To find the actual code words we can start from the leftmost word, which we assign all 0's. (here 000)

Then we proceed along and construct each succeeding word from the previous word by the following rule:

1. throw away any final 1's
2. convert the last 0 to a 1
3. add additional 0's if necessary to make the length of the word right.

In this case, the words would be:

000, 001, 01, 1000, 1001, 1010, 1011, 11.

Comments:

There are proofs that this method gives the best possible order preserving (prefix free) code, but they are surprisingly hard to find.

There are other things that one can show: if you have a given set of frequencies, the ordering of the blocks that makes this code have the longest length occurs if you put the rarest block first, then the most common block, then the second rarest, then second most common, then third, etc.

And doing it that way can require at most one more bit per block than the Huffman non-order preserving code solution has.

Exercises:

1. Find the best order preserving and non-order preserving code for the following block frequencies, in order. Also compute the Shannon theorem (H({p(q)})) bound for these frequencies.

1,21,3,4,5,35,5,4,3,5,98,21,14,17,32

2. The Shannon
bound will be exact if the frequencies of the two children of each node are exactly equal. For 3, and 4 blocks, find an arrangement of p(q) values that force you to deviate the most or nearly the most from the Shannon bound. (For example, for two blocks the Shannon bound approaches 0 as p(1) approaches 0. Yet any code for which p(1) is not 0 requires codeword length 1 for each block. What similar statement can be made for more blocks?